10.4 Food Delivery Events

In the previous two sections, we discussed the key classes we can use to represent a food delivery system: data classes `Restaurant`, `Customer`, `Courier`, and `Order` to represent individual entities, and a `FoodDeliverySystem` class to manage all of them. But even though the `FoodDeliverySystem` class has methods that allow us to *mutate* the state of the system, you might wonder: who is responsible for actually calling these methods?

If we were building a "real-world" app, we would need to write code that explicitly connects user actions (e.g., pressing a button on a mobile app) to these methods, and almost certainly rely on an existing software framework to do much of the "connecting" for us.

The approach we're taking in this chapter is a bit different. Instead of writing the code necessary to respond to real-world actions, we are going to create a *simulation* that uses a combination of preset and random data to simulate these kinds of real-world actions. The driving force of our simulation will be *events* that cause our system to mutate. For example, a "new order" event for when a customer places an order, and a "complete order" event for when a courier has delivered an order to a customer.

The `Event` interface

There are many other events we might add to the simulation, but they clearly have something in common: they are events that cause the state of the simulation to change. In 9.8 Defining a Shared Public Interface with Inheritance, we learned how to define an abstract class to represent a shared public interface, and used inheritance to relate this abstract class to concrete subclasses that must adhere to this interface. In our case, we'll define abstract `Event` class with subclasses `NewOrderEvent` and `CompleteOrderEvent` to represent different kinds of events.

Here is an initial definition of this `Event` interface. The class has one abstract method, `handle_event`, which is how we connect each event to a change in the food delivery system.

```
class Event:
    """An abstract class representing an event in a food delivery simulation.
    """

    def handle_event(self, system: FoodDeliverySystem) -> None:
        """Mutate the given food delivery system to process this event."""
        raise NotImplementedError
```

Each `Event` subclass is responsible for implementing `handle_event` based on the type of change the subclass represents. For example, the `NewOrderEvent.handle_event` method should, well, add a new order to the system. In order to implement `handle_event`, each subclass will probably need its own set of instance attributes to represent the details of the event (e.g., what order to add in a `NewOrderEvent`).

But before we discuss these subclass-specific attributes, we'll take a brief detour we'll introduce another feature of inheritance: shared instance attributes. Specifically, our simulation will need to know exactly *when* every event should happen, which every event object needs to keep track of.

Common instance attributes

We have seen that an abstract superclass declare methods that all its subclasses need to have in common, establishing a shared public interface. A superclass can also declare *public instance attributes* that its subclasses must have in common. For our `Event` class, we can establish that all event subclasses will have a timestamp indicating when the event took place. This `timestamp` attribute becomes part of the shared public interface of each subclass.

```
import datetime


class Event:
    """An abstract class representing an event in a food delivery simulation.

    Instance Attributes:
        - timestamp: the start time of the event
    """
    timestamp: datetime.datetime
```

Even though abstract classes should not be instantiated directly, we provide an initializer to initialize the common attributes (namely, `timestamp`):

```
import datetime


class Event:
    """An abstract class representing an event in a food delivery simulation.

    Instance Attributes:
        - timestamp: the start time of the event
    """
    timestamp: datetime.datetime

    def __init__(self, timestamp: datetime.datetime) -> None:
```

```
        """Initialize this event with the given timestamp."""
        self.timestamp = timestamp
```

Now let's create a new class that inherits from `Event`:

```
class NewOrderEvent(Event):
    """An event where a customer places an order for a restaurant."""
```

Remember that subclasses will inherit all the methods from their superclass. So when we attempt to initialize a `NewOrderEvent`, the Python interpreter will call `Event.__init__` (because `NewOrderEvent` did not override the parent's `__init__` method). This means we *must* provide a `datetime.datetime` object as the first argument when creating a new `NewOrderEvent` object:

```
>>> e = NewOrderEvent()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: __init__() missing 1 required positional argument: 'timestamp'
>>> e = NewOrderEvent(datetime.datetime(2020, 9, 8))
>>> e.timestamp
datetime.datetime(2020, 7, 20, 0, 0)
```

Subclass-specific attributes

It is possible that subclasses need their own attributes in addition to the ones that are common through the base class. In these scenarios, we should document our new attributes in the subclass itself. We often make these attributes private, to avoid changing the public interface declared by the abstract superclass. We do *not* need to repeat the documentation for the `timestamp` attribute; our expectation is that users should read the documentation of both the `NewOrderEvent` and `Event` classes to get the full picture of how `NewOrderEvent` is used.

```
class NewOrderEvent(Event):
    """An event representing a when a customer places an order at a
        restaurant."""
    # Private Instance Attributes:
    #   _order: the new order to be added to the FoodDeliverySystem
    _order: Order
```

To initialize this new attribute, we must define a separate initializer for `NewOrderEvent`. Here is our first attempt:

```
class NewOrderEvent(Event):
    """An event representing a when a customer places an order at a
         restaurant."""
    # Private Instance Attributes:
    #   _order: the new order to be added to the FoodDeliverySystem
    _order: Order

    def __init__(self, order: Order) -> None:
        """Initialize a NewOrderEvent for the given order."""
        self._order = order
```

This code looks correct, but has a subtle bug. By defining our own initializer for `NewOrderEvent`, we have overridden the `Event.__init__` method. Python will no longer call `Event.__init__` when creating a new `NewOrderEvent` object. However, this is problematic because **subclasses inherit methods, not attributes.** This means that the public instance attribute `timestamp` is missing from our `NewOrderEvent` object:

```
>>> order = ... # Assume we've defined an Order object here
>>> event = NewOrderEvent(order)
>>> event.timestamp
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'NewOrderEvent' object has no attribute 'timestamp'
```

So how do we make `NewOrderEvent` have both an `_order` and `timestamp` attribute? We need to modify its initializer, since it is the responsibility of the initializer to give values to all instance attributes.

First, what *should* the value of the event's `timestamp` be? A natural choice is that it should be the time that the order was placed—its `start_time` attribute. Here is our second attempt at the `NewOrderEvent.__init__` method:

```
class NewOrderEvent(Event):
    def __init__(self, order: Order) -> None:
        self.timestamp = order.start_time
        self._order = order
```

However, initializing the `timestamp` attribute directly in the subclass is bad design; code has been duplicated and that makes our code smell bad. Every time we modify the `Event` class to include new shared attributes, we'd also need to modify `NewOrderEvent.__init__` (and the initializers of every other subclass) to initialize those attributes.

So instead, we modify `NewOrderEvent.__init__` so that it directly calls `Event.__init__`.[1] Here is our third and final version of this initializer:

[1] Remember that when we call a method using the `<Class>.<method>` name, we need to pass in the `self` argument explicitly.

```
class NewOrderEvent(Event):
    """An event where a customer places an order for a restaurant."""
    _order: Order

    def __init__(self, order: Order) -> None:
        Event.__init__(self, order.start_time)
        self._order = order
```

Now, whenever we call `NewOrderEvent.__init__`, Python also calls `Event.__init__`. This causes all shared instance attributes from `Event` to be "inherited" by the `NewOrderEvent` subclass.

To summarize, we must follow two rules when inheriting from a class that defines its own initializer:

1. The initializer of a subclass must call the initializer of its superclass to initialize all common attributes.
2. The initializer of a subclass is responsible for initializing any additional attributes that are specific to that subclass.

Implementing `NewOrderEvent.handle_event`

Next, we'll show how to complete the implementation of `NewOrderEvent` by implementing its `handle_event` method. Our first attempt is quite simple, taking advantage of the methods we defined in 10.3 A "Manager" Class.

```
class NewOrderEvent(Event):
    """An event where a customer places an order for a restaurant."""
    _order: Order

    def __init__(self, order: Order) -> None:
        Event.__init__(self, timestamp)
        self._order = order

    def handle_event(self, system: FoodDeliverySystem) -> None:
        """Mutate system by placing an order."""
        system.place_order(self._order)
```

Now, there's a subtle problem with this method that we'll return to at the end of this section. A good exercise is to pause here and try to think about what the problem might be.

Implementing other `Event` subclass

Below, we've shown the implementation of our `CompleteOrderEvent`, which is quite similar to `newOrderEvent`. The major difference is that its initializer takes an explicit `datetime.datetime` argument to represent when the given order is completed.[2]

[2] By convention, the `timestamp` parameter is the first parameter, so that the subsequent parameters are seen as additional parameters needed by `NewOrderEvent` rather than `Event`. This example shows that initializers of subclasses can have different signatures than the initializer of their parent class.

```
class CompleteOrderEvent(Event):
    """When an order is delivered to a customer by a courier."""
    # Private Instance Attributes:
    #   _order: the order to be completed by this event
    _order: Order

    def __init__(self, timestamp: datetime.datetime, order: Order) -> None:
        Event.__init__(self, timestamp)
        self._order = order

    def handle_event(self, system: FoodDeliverySystem) -> None:
        """Mutate the system by recording that the order has been delivered
         to the customer."""
        system.complete_order(self._order, self.timestamp)
```

Event generation

We started off this section by asking, "when are the `FoodDeliverySystem` methods called"? We said that our simulation would have `Event` instances that would be responsible for calling these methods. But this really just changes the direction of our original question—it now becomes, "when are the `Event` instances created?"

One possible approach is to randomly create a whole set of events at the start of our simulation, and then process each of those events (in order of their `timestamp`). This approach works when the events are fairly simple and can be predictably generated all at once. However, one key feature of events in general is that *processing one event can cause other events to occur*. For example, when we process a `NewOrderEvent`, we expect that at some point in the future, a corresponding `CompleteOrderEvent` will occur.[3]

[3] Once the delivery is started, it completes. This doesn't necessarily always happen in real life, but we'll assume it does for the purposes of this case study.

To model this behaviour, we change the return type of `handle_event` from `None` to `list[Event]`, where the return value is a list of the events *caused* by the current event.

```
class Event:
    ...
```

```
    def handle_event(self, system: FoodDeliverySystem) -> list[Event]:
        """Mutate the given food delivery system to process this event.

        Return a new list of new events created by processing this event.
        """
        raise NotImplementedError
```

Here's how we might change the `NewOrderEvent` to return a `CompleteOrderEvent` at some point in the future.

```
class NewOrderEvent(Event):
    ...

    def handle_event(self, system: FoodDeliverySystem) -> list[Event]:
        """Mutate system by placing an order."""
        system.place_order(self._order)

        # Create a new CompleteOrderEvent. Right now the completion time is
        # hard-coded as 10 minutes from the order creation.
        # How could be make this more realistic by taking into account the
        # positions of the courier, customer, and restaurant?
        completion_time = self.timestamp + datetime.timedelta(minutes=10)
        return [CompleteOrderEvent(completion_time, self._order)]
```

So for every `NewOrderEvent` that is handled by our simulation, a subsequent `CompleteOrderEvent` will be handled at some point in the future.

Now here's where the problem we mentioned earlier comes in! Remember our docstring for `FoodDeliverySystem.place_order`: we cannot place an order if there are no available couriers! So what should this event do if `system.place_order` returns `False`? At the very least, in this case no `CompleteOrderEvent` should be returned.

One approach we might take is a *polling technique*, where we return a duplicate of the event to try again a little bit later. Here is our second version of this method:

```
class NewOrderEvent(Event):
    ...

    def handle_event(self, system: FoodDeliverySystem) -> list[Event]:
        """Mutate system by placing an order."""
        success = system.place_order(self._order)

        if success:
            completion_time = self.timestamp + datetime.timedelta(minutes=10)
            return [CompleteOrderEvent(completion_time, self._order)]
        else:
```

```
            self._order.start_time = self.timestamp +
        datetime.timedelta(minutes=5)
            return [NewOrderEvent(self._order)]
```

Returning no events

Our `CompleteOrderEvent` does not cause any new events to happen:

```
class CompleteOrderEvent(Event):
    ...

    def handle_event(self, system: FoodDeliverySystem) -> list[Event]:
        """Mutate the system by recording that the order has been delivered
         to the customer."""
        system.complete_order(self._order, self._timestamp)
        return []
```

A new event type

Lastly, we'll sketch one new type of event which is more conceptual, but that illustrates the power of this `Event` interface. This event type will represent a *random generation of new orders over a given time period*, which we'll use to drive our simulation.

```
class GenerateOrdersEvent(Event):
    """An event that causes a random generation of new orders.

    Private Representation Invariants:
        - self._duration > 0
    """
    # Private Instance Attributes:
    #   - _duration: the number of hours to generate orders for
    _duration: int

    def __init__(self, timestamp: datetime.datetime, duration: int) -> None:
        """Initialize this event with timestamp and the duration in hours.

        Preconditions:
            - duration > 0
        """

    def handle_event(self, system: FoodDeliverySystem) -> list[Event]:
        """Generate new orders for this event's timestamp and duration."""
        events = []

        while ...:

            new_order_event = ...  # Create a randomly-generated
        NewOrderEvent
```

```
            events.append(new_order_event)

        return events
```

We'll discuss how we might implement this class in lecture, but it's a good exercise to try to implement it yourself. There's many ways to randomly generate new events, so don't be afraid to experiment!

From events to a simulation

In this section, we focused only on defining individual `Event` classes to represent different events in our simulation. In the next section, we'll put together everything we've covered up to this point to finally get a full simulation up and running, so keep reading!